ELSEVIER

Discrete Applied Mathematics 90 (1999) 195–221

DISCRETE APPLIED MATHEMATICS

A graph theoretical approach for the yield enhancement of reconfigurable VLSI/WSI arrays ☆

Jagannathan Narasimhan [a,*], Kazuo Nakajima [b], Chong S. Rim [c]

[a] *IBM Research, T. J. Watson Research Center, P. O. Box 218, Yorktown Heights, NY 10598, USA*
[b] *Department of Electrical Engineering, University of Maryland, College Park, MD 20742, USA*
[c] *Department of Computer Science, Sogang University, Seoul, Republic of Korea, South Korea*

Received 1 November 1996; revised 1 November 1997; accepted 11 February 1998

Abstract

In this paper, we consider the yield enhancement of programmable structures by logical restructuring of the circuit placement. In this approach, an initial placement of a circuit on the array is first obtained by simulated annealing on a defect-free array. To implement the circuit on a defective array, the initial placement is reconfigured so that only the defect-free portion of the array is used. Customizing a given initial placement for each defective chip by logical restructuring, if done very fast, would be a cost effective method for yield enhancement. We describe a formulation of the circuit reconfiguration problem in terms of graphs and pebbles, wherein each processing element (PE) of the array is represented by a vertex which is classified as either defective or nondefective, depending upon whether the PE that it represents is defective or nondefective. Vertices representing PEs that are physically adjacent are connected by an edge, whose length is a measure of the proximity of the PEs. The logic elements of a circuit are represented by weighted pebbles. The initial placement of the circuit on the array corresponds to an initial placement of the pebbles on the vertices of the graph, with at most one pebble per vertex. The problem is to successively shift these pebbles along paths in the graph, such that after reconfiguration no pebble is located on a defective vertex, and an associated cost function is minimized. We describe four cost measures using weighted displacement and weighted shift of the pebbles. After presenting exact algorithms for some special cases of the problem, we prove the NP-completeness of the general cases of the corresponding decision problems.

Keywords: Reconfiguration; Yield enhancement; Placement; Polynomial time algorithms; NP-completeness

☆ This work was supported in part by National Science Foundation Grants MIP-84-51510 and CDR-88-03012 (Engineering Research Centers Program) and in part by grants from AT&T and Nissan Motor Co., Ltd.

* Corresponding author. Fax: (914) 945-4469; E-mail: jagan@watson.ibm.com.

0166-218X/99/$ – see front matter
PII: S0166-218X(97)00076-6

1. Introduction

Programmable arrays offer an efficient means of fabricating application specific integrated circuits (ASICs) with a fast turn around time. Each processing element (PE) of such an array is capable of performing the tasks required by any logic element (LE) of a circuit that is to be implemented on it. The PE may be either a simple gate as in the case of programmable gate arrays (PGAs) or a processor in more complex arrays, which we refer to as programmable processor arrays (PPAs) for uniformity. To implement a circuit on an array, each of its LEs is first mapped onto a PE of the array. The array is then programmed by making or breaking contacts between the terminals of the PEs and the wires provided in the routing channels, and between those wires, in accordance with the connection requirement of the circuit [10].

Kumar et al. [5] have suggested a method to enhance the yield of PGAs by exploiting the functional interchangeability of ther PEs so that circuits are implemented only on the nondefective gates of the PGA. In their approach an initial placement of a circuit is obtained on an ideal programmable array; that is, the one free of defects, using some time-consuming process like simulated annealing [4]. Note that it is not suitable to use an expensive process such as that used for obtaining the initial placement to place each defective array. The circuit is then implemented on an actual array with the same architecture as the defect-free ideal array but with defective PEs. For this purpose the initial placement is reconfigured using some very fast scheme so that the new placement uses only nondefective PEs. Clearly, the circuit is reconfigurable only if the number of defective PEs that implement LEs is not greater than the number of nondefective PEs that do not implement LEs. Utilization of preprocessed defective arrays instead of discarding them is an effective means of enhancing their yield.

Central to the yield enhancement scheme just outlined is the strategy for reconfiguration. With a very fast reconfiguration strategy this yield enhancement system may be used for the on-line layout of a circuit on each individual defective chip. This problem of reconfiguring programmable arrays has quite recently been formulated in terms of a graph and pebbles and called the *pebble shift problem* by Narasimhan et al. [8]. Using algorithms based on the pebble shift problem, Narasimhan et al. have demonstrated that the yield enhancement scheme proposed by Kumar et al. can indeed be of great practical use.

In the formulation of the pebble shift problem a programmable array is represented by a graph $G=(V,E)$ which is constructed in the following way. Each PE of an array is represented by a vertex $v \in V$. For each pair of adjacent PEs an edge of an appropriate length is drawn between the corresponding vertices. Different edge lengths may arise due to various situations; for example, there may be wide routing channels between some pairs of adjacent PEs. Each vertex is classified as *defective* or *nondefective* depending upon whether its corresponding PE is defective or nondefective, respectively. Each LE of a circuit to be implemented on the array is regarded as a pebble. Depending

on the number of terminals of an LE that are connected to nets, the corresponding pebble is assigned an integer weight.

The mapping of an LE onto a PE is equivalent to the placement of the pebble representing the LE on the vertex corresponding to the PE. The weight of a pebble is a measure of the effort required to move the pebble by a unit distance from its current position to its adjacent position. With respect to circuit reconfiguration, an LE that is connected to very few nets may be more amenable to relocation as compared to one connected to many nets.

In the initial placement, some pebbles may be located on defective vertices, indicating that the corresponding LEs are implemented by defective PEs. Reconfiguring the placement of the circuit so that it is implemented only on nondefective PEs is analogous to changing the placement of the pebbles on the graph in such a way that all the pebbles are located on nondefective vertices. Furthermore, to ensure that the characteristics of the initial placement are maintained as much as possible, each pebble that is located on a defective vertex, say d, is to be moved to a nondefective vertex, by successively shifting all the pebbles along a path from d to a vacant and nondefective vertex. A logical remapping of the LEs on a chip may be seen as a sequence of such shifts of pebbles on the graph that leaves no pebble on a defective vertex and that minimizes an associated cost function defined for the problem. Such a shifting mechanism is used so as to retain the relative ordering of the pebbles in the initial placement as much as possible.

In a formulation of the problem by Narasimhan et al. [8] the *total displacement* of the pebbles on the graph was used as the cost measure. More precisely the cost was based upon the sum of the weight of each pebble multiplied by the distance between its initial location and its final location. Three other cost measures, namely *maximum displacement*, *total shift* and *maximum shift* were also mentioned in their formulation.

After presenting some graph theoretical preliminaries we formally state the pebble shift problem in Section 2 in terms of costs based on displacement. In Section 3 we present fast exact algorithms for the case in which the graph is a path or cycle and the pebbles are of arbitrary weight, and for the case in which the graph is arbitrary and all the pebbles are of the same weight. The first case corresponds to the reconfiguration of I/O buffers of a programmable chip. The algorithms for the second case have been shown to be efficient heuristics for reconfiguring the PEs on a defective PGA. We then analyze the computational complexity of the pebble shift problem under all the four cost measures in Section 4. More specifically, we prove that the total displacement minimization problem for undirected as well as directed graphs and the maximum displacement minimization problem for directed acyclic graphs are NP-hard even when all edges are of the same length and all pebbles are of weight 1 or 2. These results can be easily applied to costs based on shift as well.

Before presenting our results, we review some graph theoretical terms and formally define the pebble shift problem under various cost functions in the next section.

2. Preliminaries

Let $G=(V,E)$ be an undirected (or directed) graph with nonempty vertex set V and nonempty edge set E. A *path* (or *directed path*) from vertex s to vertex t is a sequence $[s=v_1, v_2, \ldots, v_k=t]$ of distinct vertices such that $(v_i, v_{i+1}) \in E$ for $i=1,2,\ldots,k-1$. Let a nonnegative integer $l(u,v)$ denote the length of each edge $(u,v) \in E$. The length of the path is given by $\sum_{i=1}^{k-1} l(v_i, v_{i+1})$. A *shortest path* between two vertices u and v is a path from u to v with the least length. We denote the length of such a shortest path by $d(u,v)$. A *cycle* (or *directed cycle*) is a sequence $[v_1, v_2, \ldots, v_{k-1}, v_k, v_1]$ in which $v_1, v_2, \ldots, v_{k-1}$, and v_k are distinct vertices such that $(v_i, v_{i+1}) \in E$ for $i=1,2,\ldots,k-1$ and $(v_k, v_1) \in E$. We refer to a path and a cycle in which all the vertices are distinct as a *simple path* and a *simple cycle*, respectively. If a directed graph $G=(V,E)$ has no cycle, it is called an *directed acyclic graph*.

A graph $G'=(V',E')$ is called a *subgraph* of G if $V' \subseteq V$ and $E' \subseteq E$. A graph $G=(V,E)$ is called a *connected graph* if the vertex set V cannot be partitioned into two sets V_1 and V_2 such that there is no edge $(v_1,v_2) \in E$ with $v_1 \in V_1$ and $v_2 \in V_2$. A graph $G=(V,E)$ is called a *planar graph* if it can be drawn in a plane in such a way that (1) each vertex in V is represented by a point, (2) each edge $(u,v) \in E$ is represented by a continuous line connecting the two points that represent u and v, and (3) no two lines that represent edges share any points, other than their end points. Such a drawing is called a *planar embedding* of G.

Let $X=\{x_1,x_2,\ldots,x_a\}$ and $Y=\{y_1,y_2,\ldots,y_b\}$ be the coordinates of points located on the orthogonal axes of a plane such that $x_{i+1} > x_i$ for $i=1,\ldots,a-1$ and $y_{i+1} > y_i$ for $i=1,\ldots,b-1$. Let $G=(V,E)$ be a graph where each vertex $v_{ij} \in V$ represents a point (x_i, y_j) with $x_i \in X$ and $y_j \in Y$ and $(v_{ij}, v_{kl}) \in E$ if and only if $|i-k|+|l-j|=1$. G is called a *grid graph*. Each edge of such a graph is parallel to one of the two orthogonal axes. The *Manhattan distance* between two vertices v_{ij} and v_{kl} in G is defined as $|x_i - x_k| + |y_j - y_l|$. A path from v_{ij} to v_{kl} in G whose length is the Manhattan distance is called a *shortest Manhattan path* between v_{ij} and v_{kl}.

We now review terms relevant to the pebble shift problem [8]. Let $G=(V,E)$ be a graph that represents a programmable array, P be a set of pebbles such that $|P| \leqslant |V|$, and let $w(p)$ denote the weight of the pebble p. Initially, all pebbles are placed on G with at most one pebble on a vertex. This placement is called the *initial placement* and is denoted by C_1. A vertex v is said to be *occupied* if a pebble is placed on it and *vacant* otherwise. In the former case let $p(v)$ denote the pebble placed on v. Let D_1 and F_1 be the sets of occupied and defective and of vacant and nondefective vertices, respectively, in the graph. Fig. 1 shows a 4×4 programmable array and Fig. 2 shows the corresponding graph with an initial placement of pebbles. For this undirected "4×4 grid" graph with 16 vertices and 24 edges, $D_1=\{v_{12}, v_{14}, v_{21}\}$ and $F_1=\{v_{23}, v_{24}, v_{41}, v_{43}\}$.

A path $[d=v_1, v_2, \ldots, v_l=f]$ is called a *reconfiguration path* and denoted by $Q(d,f)$, if d is occupied and defective, f is vacant and nondefective, and for $i=2,\ldots,l-1$, v_i is either vacant and defective or occupied. Note that if $l=2$, $Q(d,f)$ is simply the

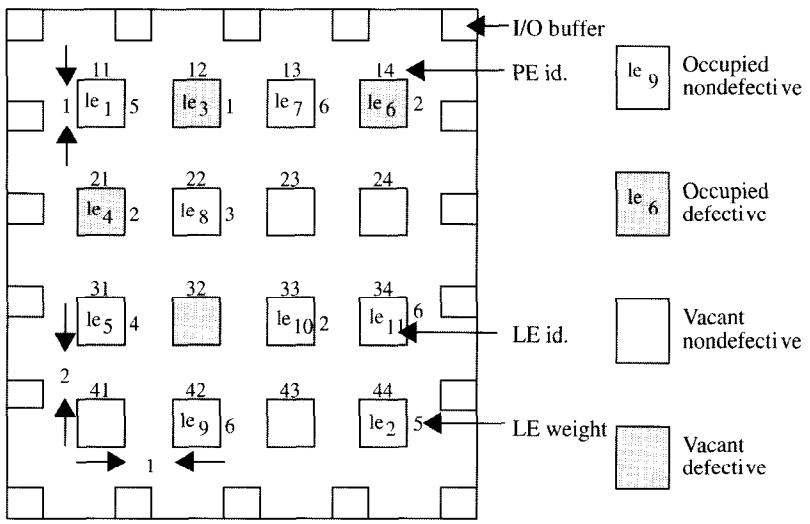

Fig. 1. A programmable array with a circuit implemented on it.

edge $(d,f) \in E$. For such a reconfiguration path $Q(d,f)$, let $w_1, w_2, \ldots, w_r$ be the occupied vertices that appear in this order along the path and let $w_0 = d$ and $w_{r+1} = f$. We define an operation called the *reconfiguration* along $Q(d,f)$ to be the successive shifting of each pebble $p(w_i)$ from w_i to w_{i+1} for $i = r, r-1, \ldots, 0$. A reconfiguration reduces $|D_1|$ and $|F_1|$ each by one, yielding a new placement C_2. The shift of a pebble $p \in P$ due to Reconfiguring along $Q(d,f)$ is denoted by $C(p.Q)$ and is defined as follows: if $p = p(w_i)$ where $w_i \in \{w_1, w_2, \ldots, w_r\}$ then $C(p,Q) = w(p) \times d(w_i, w_{i+1})$, otherwise $C(p,Q) = 0$. The shift due to reconfiguring along the path $Q(d,f)$ is defined as $\sum_{i=1}^{i=r} C(p(w_i),Q)$ and is denoted by $C(d,f)$. Throughout this paper, let $m = |D_1|$ and $n = |V|$. A *reconfiguration sequence* R is an ordered sequence of reconfiguration paths $Q(d_i,f_i)$ for $i = 1,2,\ldots,m$, which when applied successively to the placement C_1 will result in a final placement C_{m+1} with no occupied and defective vertex.

Let $v^1(p)$ and $v^{m+1}(p)$ denote the vertices on which a pebble p is located in placements C_1 and C_{m+1}, respectively. The *total displacement* due to the application of the reconfiguration sequence R is given by $\sum_{p \in P} w(p) \times d(v^1(p), v^{m+1}(p))$ and is denoted by $TD(R)$. The *maximum displacement* of the reconfiguration sequence R, denoted by $MD(R)$, is given by $\max_{p \in P}\{w(p) \times d(v^1(p), v^{m+1}(p))\}$. In Fig. 2 the reconfiguration sequence $R = \{Q(v_{21}, v_{23}), Q(v_{12}, v_{43}), Q(v_{14}, v_{24})\}$ is shown using bold lines. Based on a horizontal routing capacity of 2 and a vertical routing capacity of 1 the distances between adjacent vertices in the horizontal and vertical directions are estimated as 3 units and 2 units, respectively. Fig. 3 shows the final placement obtained using this sequence, where $TD(R) = 2 \times (2+3+2) + 1 \times 3 + 3 \times 2 + 2 \times 3 + 2 \times 3 = 35$. and $MD(R) = \max\{2 \times (2+3+2), 1 \times 3, 3 \times 2, 2 \times 3, 2 \times 3\} = 14$. It should be noted that

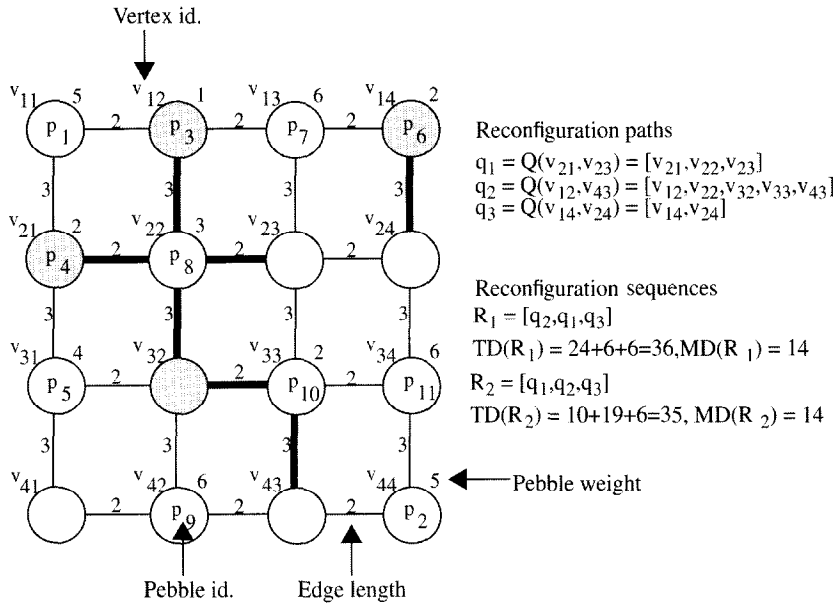

Fig. 2. The graph for the programmable array in Fig. 1.

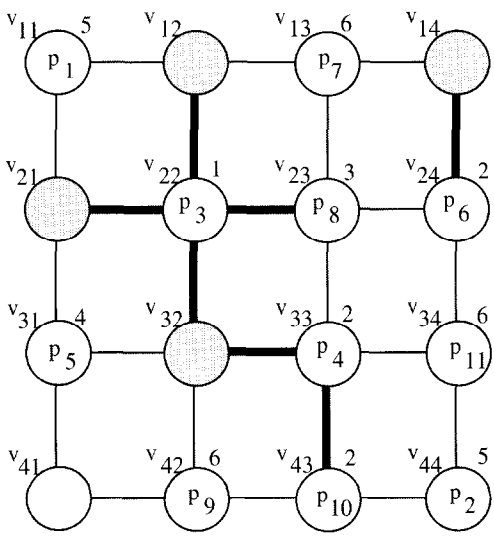

Fig. 3. The final placement after applying R_2 to the placement of Fig. 2.

the total displacement and the maximum displacement are determined by the choice of the paths as well as the order in which they are applied.

The total shift of a pebble p due to the application of R is defined as $\sum_{Q(d,f)\in R} C(d,f)$ and denoted by $s(p)$. Note that while a pebble may be shifted a lot during reconfiguration, it may be displaced very little, or even not be displaced at all from

its original position after the application of the entire reconfiguration sequence R. The *total shift* due to the application of R is given by $\sum_{p\in P} s(p)$ and is denoted by $TS(R)$. Using the same analogy, the *maximum shift* due to the application of R is given by $\max_{p\in P} s(p)$ and is denoted by $MS(R)$.

With these preliminary definitions we formally define new versions of the pebble shift problem mentioned earlier. The first version uses the total displacement and the second the maximum displacement as the cost functions, respectively. The decision versions of these problems are given for both directed as well as undirected graphs.

Total Displacement Problem (*TDP*)

Instance: A graph $G=(V,E)$, a set P of pebbles, an initial placement of the pebbles on G, with a set D_1 of occupied and defective vertices and a set F_1 of vacant and nondefective vertices, and an integer c.

Question: Is there a reconfiguration sequence R with $TD(R)\leqslant c$?

Maximum Displacement Problem (*MDP*)

Instance: Same as the instance for *TDP*.

Question: Is there a reconfiguration sequence R with $MD(R)\leqslant c$?

The optimization versions of the total and maximum displacement problems are to find reconfiguration sequences R, with the least values for $TD(R)$ and $MD(R)$, respectively. Such sequences are said to be *optimal*. Using cost measures based on *shifts* as defined earlier, the *total shift problem* (*TSP*) and *maximum shift problem* (*MSP*) can be defined similar to *TDP* and *MDP*, respectively.

3. Polynomial-time exact algorithms

3.1. Pebble shift on a cycle

In this section we assume that the graph $G=(V,E)$ is a path or cycle. Note that this type of problem corresponds to the problem of reconfiguration of I/O buffers located on the periphery of an array [8]. We provide an $O(m^2n)$ time exact algorithm for the optimization version of *TDP* on a path. Since this algorithm is identical to the algorithm for minimizing the total shift presented by Narasimhan et al. [7] we have omitted the proof of its correctness. Next, we present an $O(mn)$ time exact algorithm for the optimization version of *MDP* with pebbles are of arbitrary weight. The algorithms for the optimization version of *TDP* and *MDP*, with minor modifications, can also be applied to the case when $G=(V,E)$ is a cycle.

Let $G=(V,E)$ be a path $[v_1, v_2,\ldots,v_n]$. Sequences of distinct vertices $[v_j, v_{j+1},\ldots, v_{k-1}, v_k]$ and $[v_j, v_{j-1},\ldots, v_{k+1}, v_k]$ are called a *right* and *left path* from vertex v_j to vertex v_k, and are denoted by $Q^{\mathrm{r}}(v_j, v_k)$ and $Q^{\mathrm{l}}(v_j, v_k)$, respectively, where the superscripts r and l denote the right and left directions, respectively. If they represent reconfiguration paths, such paths are called a *right* and *left reconfiguration path*, respectively. The vertices v_j and v_k are called the *end vertices* of both paths. The length

of a path $Q^x(v_j, v_k)$, where x represents r or l, is defined as $\sum_{i=j}^{k-1} l(v_i, v_{i+1})$ if $x = r$ and $\sum_{i=k}^{j-1} l(v_i, v_{i+1})$ if $x = l$ and is denoted by $a^x(v_j, v_k)$. This is in fact the distance between v_j and v_k measured from v_j to v_k in the direction x.

Let the set D_1 of occupied and defective vertices in the initial placement be $\{d_1, d_2, \ldots, d_m\}$. We assume that if d_k and d_{k+1} correspond to the vertices v_i and v_j, respectively, then $i < j$. Let $P_D = [pd_1, pd_2, \ldots, pd_m]$ be a list of pebbles such that for each $i = 1, 2, \ldots, m$, $pd_i = p(d_i)$ in the initial placement.

Total Displacement. The algorithm for solving the pebble shift problem first uses Procedure *PebbleDirection* to determine the direction in which each pebble pd in P_D must be shifted in order to obtain an optimal solution. Using these directions, Procedure *PathSequence* determines a sequence of right and left reconfiguration paths that solves the pebble shift problem for the case of a path. In the algorithm $iloc(p)$ and $cloc(p)$ denote the vertices on which a pebble p is located in the initial placement and current placement, respectively, and $dir(p)$ denotes the direction in which p has been shifted from $iloc(p)$ to reach $cloc(p)$. In the initial placement $iloc(p) = cloc(p)$ and $dir(p) = NULL$.

Algorithm (*Total Shift*).

Input: A simple path $G = (V, E)$, a set of pebbles, their placement C_1 on G with a set $D_1 = \{d_1, d_2, \ldots, d_m\}$ of m occupied and defective vertices, a set F_1 of vacant and nondefective vertices, and the list $P_D = [pd_1, pd_2, \ldots, pd_m]$ of pebbles that are located on the vertices of D_1 in the initial placement.

Output: An optimal reconfiguration sequence R.

1. **if** $|D_1| > |F_1|$ **then** report failure and **exit end if**.
2. Using Procedure *PebbleDirection*, determine the direction in which each pebble pd in P_D is to be shifted.
3. Using Procedure *PathSequence* and the directions of the pebbles in P_D obtained at Step 2, generate a reconfiguration sequence R.

end *TotalDisplacement*

The core of this algorithm is Procedure *PebbleDirection* which determines the direction in which each pebble is to be shifted. This procedure works in two phases. In the first phase using Procedure *VertexAddition*, we append at the right end of the path G, m extra nondefective vertices $v_{n+1}, v_{n+2}, \ldots, v_{n+m}$ and edges (v_k, v_{k+1}) with $l(v_k, v_{k+1}) = \infty$ for $k = n, n+1, \ldots, n+m-1$. We then scan the path from left to right and shift each pebble pd in P_D to the closest possible nondefective vertex on its right. This may force other pebbles that are on the right of pd to shift to the right as well. Procedure *RightShift* is used to perform the shifting just described. Note that since $l(v_k, v_{k+1}) = \infty$ for $k = n, n+1, \ldots, n+m-1$, no pebble will stay in any of the vertices $v_{n+1}, v_{n+2}, \ldots, v_{n+m}$ in the final placement.

After Phase I is completed, there are no pebbles on defective vertices and each of the pebbles has either been moved to the right from its initial position or has not

been moved at all. We obtain an optimal solution in the second phase by iteratively improving upon the placement obtained in the first phase. We only determine the direction in which each pebble in P_D is to be shifted. An optimal reconfiguration sequence can easily be generated once these directions are known.

If a pebble that has been shifted across some edges is subsequently shifted back across the same edges, we say that it has been *retracted* across these edges. Let the two directions be denoted by *LEFT* and *RIGHT*. For $x \in \{LEFT, RIGHT\}$, we denote the direction opposite to x by $\bar{x}$.

Two procedures, *Reverse*(pd, dir) and *CostChange*(pd, dir), are used repeatedly in Phase II of Procedure *PathDirection*. If a pebble pd has been shifted to the right direction from $iloc(pd)$ to $cloc(pd)$, Procedure *Reverse*$(pd, LEFT)$ is used to actually shift pd to the closest nondefective vertex on the left of $iloc(pd)$. Note that $dir(pd) = LEFT$ after this shift. To perform such a shifting of pd it may be necessary to retract other pebbles that are to the left of pd and then possibly shift them further leftwards on to appropriate nondefective vertices. We refer to such retractions and shifts as *forced retractions* and *forced shifts*, respectively. Since changing the direction of pd requires it to be retracted and shifted further to the left, we consider its retraction and shifting to be forced. Because of the vacant vertex created by relocating pd and possibly other vertices to be vacated by the forceful retractions of other pebbles, some pebbles to the right of pd may also be retracted. It should be noted that such pebbles p' are only retracted as close to $iloc(p')$ as possible but not shifted leftwards beyond $iloc(p')$. The above relocations of pebbles incurred by the change in the direction of pd that includes the possible forced retractions and forced shifts of pebbles to the left of pd and the retractions of pebbles to the right of pd is called the *reversal* of pd.

If pd has been shifted rightwards from $iloc(pd)$ to a nondefective vertex $cloc(pd)$, Procedure *CostChange*(pd, dir) computes the change in cost due to the reversal of pd. Suppose that a pebble p is retracted from vertex u_1 to vertex u_2 due to the reversal of pd. Note that u_2 may possibly be the same as $iloc(p)$. The cost of this retraction is computed as $-w(p) \times a^{LEFT}(u_1, u_2)$. A pebble p that is not retracted but only shifted farther away from $iloc(p)$ from u_1 to u_2 is given by $+w(p) \times a^{LEFT}(u_1, u_2)$. In this case u_1 may be $iloc(p)$. It is possible that a pebble p is retracted all the way from u_1 to $u_2 = iloc(p)$ and then shifted further leftwards from u_2 to vertex u_3. The change in cost contributed by p in this case is computed as $-w(p) \times a^{LEFT}(u_1, u_2) + w(p) \times a^{LEFT}(u_2, u_3)$. Procedure *CostChange*$(pd, LEFT)$ returns the sum of the changes in cost contributed by each pebble p that is relocated by the reversal of pd. By predetermining the distance of each vertex from vertex v_1 before starting the algorithm, these values can easily be determined in constant time.

With these procedures in place we describe the Procedure *PebbleDirection* below. In its second phase we repeat the following process until there is no improvement in the cost. Let P_1 be a list of pebbles that is reset to P_D at the beginning of each *iteration* of the outer **while** loop. At each *step* of the inner **while** loop we remove the leftmost pebble p from P_1. If $dir(p)$ is *RIGHT* and *CostChange*$(p, LEFT)$ returns a negative value, we reverse p.

Procedure *PebbleDirection*

Phase I
1. *VertexAddition*(m).
2. *RightShift*().
3. *improvement* = *TRUE*.

Phase II
4. **while** *improvement* = *TRUE* **do**
   A. *improvement* = *FALSE*; $P_I = P_D$.
   B. **while** P_I is not empty **do**
      i. Remove the leftmost pebble *pd* from P_I.
      ii. **if** *dir*(*pd*) = *RIGHT* **then**
         if *CostChange*(*pd*, *LEFT*) < 0 **then**
            a. *improvement* = *TRUE*.
            b. *Reverse*(*pd*, *LEFT*).
            end if
         end if
      end while
   end while

end *PebbleDirection*.

We first illustrate the execution of the Procedure *PathDirection* with the example shown in Fig. 4(a) which depicts an initial placement of the pebbles on a path with 17 vertices, seven of which are occupied and defective. For this example $D_I = \{v_3, v_4, v_7, v_8, v_{11}, v_{12}, v_{16}\}$, $F_I = \{v_1, v_2, v_5, v_6, v_9, v_{13}, v_{14}, v_{15}, v_{17}\}$, and $P_D = [p_1, p_2, p_3, p_4, p_6, p_7, p_8]$. Note that each pebble and its weight are given inside a circle representing a vertex. The placement after the completion of Phase I is shown in Fig. 4(b). Note that no vertices appended at the right end of the path are shown in the figures since the location of the vacant and nondefective vertices in this example ensures that Procedure *RightShift* will execute successfully. In Step 1 of Iteration 1 of Phase II, the change in cost of reversing pebble p_1 is computed as *CostChange*(p_1, *LEFT*) $= 1 \times (-4-2+1) + 2 \times (-1) = -7$. Thus pebble p_1 is reversed, resulting in the placement of Fig. 4(c). In the next step *CostChange*(p_2, *LEFT*) $= 1 \times 1 + 2 \times (-4+2+1) = -1$. Pebble p_2 is reversed in this step, yielding the placement of Fig. 4(d). The changes in cost of reversing pebbles p_3, p_4, and p_6 are all positive and pebble p_7 cannot be reversed since enough vacant and nondefective vertices are not available on the left side of $d_6 = v_{12}$. The placement therefore remains unchanged during Steps 3 to 6 of Iteration 1. In Step 7 of Iteration 1, *CostChange*(p_8, *LEFT*) $= 5 \times (-1-1+5) + 1 \times (-1) + 1 \times (-1-4-1) + 3 \times (-1) + 1 \times (-1) + 3 \times (-3+1) = -2$. Pebble p_8 is reversed in this step, leading to the placement of Fig. 4(e). This completes Iteration 1.

In the first three steps of Iteration 2, the directions *dir*(p_1), *dir*(p_2), and *dir*(p_3) are *LEFT*. In Step 4 the change in cost of reversing pebble p_4 is positive. In Step 5 the *CostChange*(p_6, *LEFT*) $= 5 \times 1 + 1 \times (-1+1+5) + 1 \times 1 + 3 \times (-1-4+1)$

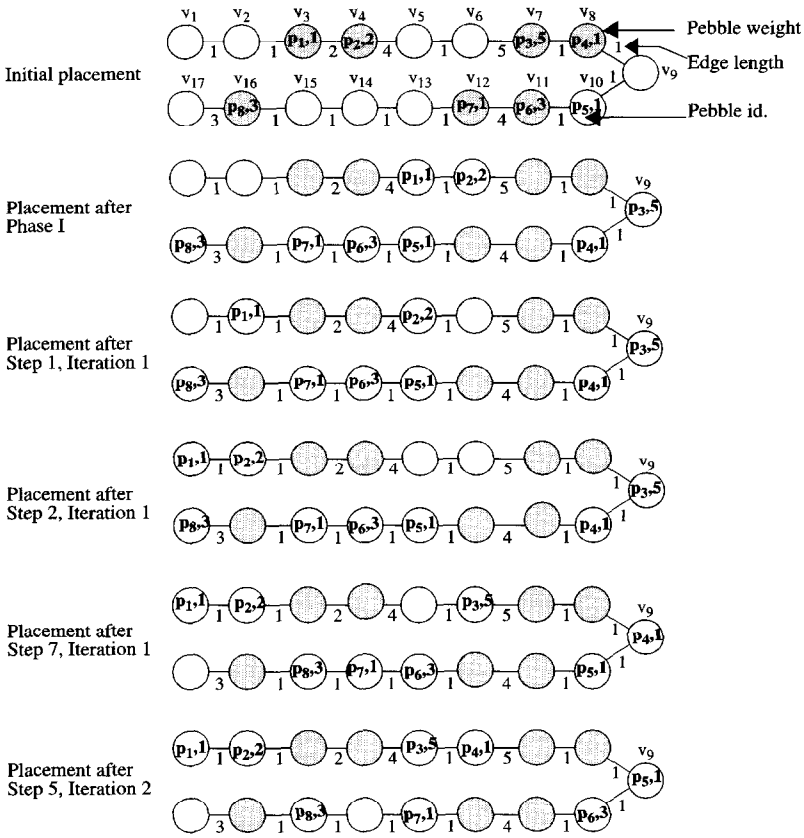

Fig. 4. Pebble shifting on a path; an example.

$+1\times(-1)=-2$. Pebble p_6 is reversed in this step, leading to the placement of Fig. 4(f). In Step 6 pebble p_7 cannot be reversed due to the nonavailability of sufficient vacant and nondefective vertices on the left side of $iloc(p_7)$. Since $dir(p_8)$ is *LEFT*, no further change in the placement occurs in this iteration.

In Iteration 3 we first note that only $dir(p_7)$ is *RIGHT* and that it cannot be reversed due to the nonavailability of sufficient vacant and nondefective vertices on the left of vertex $v_{12}=iloc(p_7)$. Since no pebble is reversed in this iteration, no more iterations are performed in Procedure *PebbleDirection*. As shown later, the directions of pebbles in P_D obtained from Procedure *PathDirection* are used by Procedure *PathSequence* to generate the minimum cost reconfiguration sequence $R=[Q^l(v_3,v_1), Q^l(v_4,v_2), Q^l(v_7,v_5), Q^l(v_8,v_6), Q^l(v_{11},v_9), Q^r(v_{12},v_{13}), Q^l(v_{16},v_{15})]$.

We now prove that Procedure *PebbleDirection* determines the directions in which the pebbles in P_D must be shifted in an optimal solution for the pebble shift problem. At the end of Phase I of the procedure any pebble that has been shifted has moved to the right and no pebble is located on a defective vertex. Furthermore, no pebble is shifted more than is necessary to obtain such a placement. In each iteration of Phase II

we scan the pebbles in P_D from left to right and attempt to systematically reverse each pebble pd in P_D with $dir(pd) = RIGHT$. The following lemma establishes a property of Phase II of Procedure *PebbleDirection*, which is used later to prove the correctness of Algorithm *PathTotal*.

Lemma 1. *At the termination of Phase II of Procedure PebbleDirection, there is no pebble pd in* P_D *with* $dir(pd) = LEFT$ *such that* $CostChange(pd, RIGHT) < 0$.

Proof. There are two nested **while** loops in Phase II of the procedure. Assuming that the following claim (A) is true, we first prove, by induction on the number of iterations, that at the end of the outer **while** loop there is no pebble pd in P_D with $dir(pd)=LEFT$ whose reversal results in a negative change in displacement.

Claim A. *If at the beginning of each execution of the inner while loop there is no pebble pd with* $dir(pd) = LEFT$ *such that* $CostChange(pd, RIGHT) < 0$, *then at the end of the execution of the loop there is no such pebble.*

At the beginning of Phase II, $dir(pd) = RIGHT$ for each pebble pd in P_D and hence there is no pebble pd with $dir(pd) = LEFT$ such that $CostChange(pd, RIGHT) < 0$. Thus, the basis for the induction is true. Suppose that at the beginning of Iteration i there is no pebble pd in P_D with $dir(pd) = LEFT$ such that $CostChange(pd, RIGHT) < 0$. Due to Claim A, there is no such pebble at the end of this iteration. By induction, we know that there is no such pebble at the end of execution of the outer **while** loop.

We now prove the above Claim A. We do this by induction on the number of steps in the inner **while** loop. The basis for the induction is true because Claim A asserts that at the beginning of execution of the inner **while** loop there is no pebble pd in P_D with $dir(pd) = LEFT$ such that $CostChange(pd, RIGHT) < 0$. For the induction hypothesis we assume that after processing pebble pd_i in the ith step, there is no pebble pd in P_D with $dir(pd) = LEFT$ such that $CostChange(pd, RIGHT) < 0$. Consider the next candidate pebble pd_{i+1} in P_D for reversal.

If $CostChange(pd_{i+1}, LEFT) > 0$, no change occurs in the placement of the pebbles. By the induction hypothesis, there is no pebble pd in P_D with $dir(pd) = LEFT$ such that $CostChange(pd, Right) < 0$ after processing pd_{i+1}.

If $CostChange(pd_{i+1}, LEFT) < 0$, we reverse pd_{i+1}. We first consider the pebbles on the right of pd_{i+1}. If any such pebble p in P has moved during this reversal, it has only been retracted. That is, p has moved in the $LEFT$ direction and closer to $iloc(p)$. If p is a pebble in P_D, $dir(p)$ must have been $RIGHT$ and remains the same. Thus, no pebble pd in P_D with $dir(pd) = LEFT$ has moved on the right of pd_{i+1}. By the induction hypothesis, the reversal of such a pebble pd does not result in a negative change in cost. Thus, at the end of the $(i+1)$st step there is no pebble pd in P_D with $dir(pd) = LEFT$ such that $CostChange(pd, LEFT) < 0$.

We now consider the pebbles on the left of pd_{i+1} including pd_{i+1} itself. Suppose that among them there is some pebble pd in P_D with $dir(pd) = LEFT$ such that

$CostChange(pd, RIGHT) < 0$ and that it is indeed reversed. The reversal of pd_{i+1} either (a) forced pd to move leftwards or (b) did not force pd to move leftwards. In the former case, the subsequent reversal of pd must move pd_{i+1} rightwards. In the latter case, suppose that the subsequent reversal of pd did not move pd_{i+1} rightwards. In such a case, reversing pd before reversing pd_{i+1} would have resulted in a negative change in cost, contradicting the induction hypothesis. Therefore, reversing pd must move pd_{i+1} rightwards.

Let C_v and C_w denote the placements of pebbles just before pd_{i+1} was reversed and just after pd was reversed, respectively. Let v and w be the vertices on which pd_{i+1} was located in the placements C_v and C_w, respectively. Furthermore, let $dv = dir(pd)$ in the placement C_v. Since reversing pd_{i+1} as well as pd in opposite directions resulted in a negative change in cost, the cost of the reconfiguration sequence associated with placement C_v must be higher than that associated with placement C_w. For convenience of notation we refer to these costs as those of the placements.

Since the reversal of pd_{i+1} shifted it to the closest nondefective vertex to the left of $iloc(pd_{i+1})$, vertex w must be on the right of $iloc(pd_{i+1})$. Since $iloc(pd)$ is defective, $dv \neq NULL$ and hence $dv = RIGHT$ or $LEFT$. We first consider the case in which $dv = RIGHT$. After its reversal pd is located on the closest nondefective vertex to the right of $iloc(pd)$ and hence vertex w cannot be to the right of v. Thus, either $w = v$ or w is located to the left of v. In either case, before the reversal of pd_{i+1} there must be a pebble pd_j with $j \leqslant i$ such that $dir(pd_j) = RIGHT$ and $CostChange(pd_j, LEFT) < 0$ but it was not reversed. Since the algorithm scans the pebbles from left to right, pd_j must have been reversed, a contradiction.

We now consider the case in which $dv = LEFT$. Note that after the reversal of pd_{i+1}, $CostChange(pd, RIGHT) < 0$. This is possible only if before the reversal of pd_{i+1} either (a) $CostChange(pd, RIGHT) < 0$, or (b) there was a pebble pd_j with $j \leqslant i$ such that $CostChange(pd_j, LEFT) < 0$ but it was not reversed. The former case contradicts the induction hypothesis. In the latter case, since the algorithm scans the pebbles from left to right, pd_j must have been reversed before the reversal of pd_{i+1}, a contradiction.

We can therefore conclude that there is no pebble pd in P_D on the left of pd_{i+1} including pd_{i+1} itself with $dir(pd) = LEFT$ such that $CostChange(pd, RIGHT) < 0$. In summary, if $CostChange(pd_{i+1}, LEFT) < 0$, there is no pebble pd in P_D such that $dir(pd) = LEFT$ and $CostChange(pd, RIGHT) < 0$ at the end of the $(i+1)$st step. This completes the proof of the lemma. □

Remark 4.1. Note that Procedure *PebbleDirection* terminates when there is no pebble pd in P_D with $dir(pd) = RIGHT$ such that $CostChange(pd, LEFT) < 0$. Therefore, at the termination of the procedure, there is no pebble pd in P_D whose reversal results in a negative change in cost, whether $dir(pd) = LEFT$ or $RIGHT$.

Theorem 3. *Algorithm PathTotal determines correctly, in $O(m^2 n)$ time, an optimal solution for the pebble shift problem for the case of a path.*

Proof. Suppose that there is a reconfiguration sequence for a given instance of the pebble shift problem whose cost is less than that of the reconfiguration sequence R obtained by the algorithm. We show that in the final placement C_R obtained by applying R, there is at least one pebble in P_D whose reversal results in a negative change in cost. Since R is not the least costly solution, there is at least one pebble p in P whose shifting to a nondefective vertex, denoted by $cloc'(p)$ that is closer to $iloc(p)$ than $cloc(p)$, will reduce the total cost of shifting. This shift would cause other pebbles q to move from $cloc(q)$ to a new location $cloc'(q)$. Let $dir(q)$ and $dir'(q)$ be the directions of shifting q to the vertices $cloc(q)$ and $cloc'(q)$, respectively, from $iloc(q)$.

In the following, we assume, without loss of generality, that $dir(p) = LEFT$. A similar line of reasoning can be used when $dir(p) = RIGHT$. We first note that in the placement C_R, p was shifted leftward due to the leftward shifting of a pebble pd_i, which is either p itself if it is in P_D or the closest pebble in P_D to the right of p. Shifting p from $cloc(p)$ to $cloc'(p)$ will shift pd_i rightwards in the placement C_R. If $dir'(pd_i) = dir(pd_i) = LEFT$, pebble pd_i was shifted to $cloc(pd_i)$ due to the leftward movement of pd_{i+1} in P_D to $cloc(pd_{i+1})$ in the placement C_R. If $dir'(pd_{i+1}) = dir(pd_{i+1}) = LEFT$, pebble pd_{i+1} was shifted to $cloc(pd_{i+1})$ due to the leftward movement of pd_{i+2} in P_D to $cloc(pd_{i+2})$ in the placement C_R. Proceeding rightwards in this way, we can find at least one pebble pd_k in P_D with $k \geqslant i$ such that $dir(pd_k) = LEFT$ and $dir'(pd_k) = RIGHT$.

Shifting p from $cloc(p)$ to $cloc'(p)$ creates vacant and nondefective vertices between these two vertices onto which pebbles to the left of p could possibly be shifted to reduce the cost. Shifting these pebbles would in turn create other vacant and nondefective vertices onto which pebbles further to the left could be shifted to reduce the total cost. Therefore, by shifting some pebbles that are to the right and possibly to the left of p including p rightwards in the placement C_R, we obtain a new placement $C_{R'}$ that costs less than C_R. Such a movement of pebbles corresponds to the reversal of a single pebble pd in P_D or a combination of the reversals of several pebbles pd in P_D such that $dir(pd) = LEFT$ before the reversal. In the latter case, some of these reversals may increase the cost whereas others reduce the cost. Since $C_{R'}$ costs less than C_R, at least one of these reversals must result in a negative change in cost. Among the pebbles in P_D that are reversed, let pd' be the first pebble in the order of reversal whose reversal results in a placement with cost less than that of C_R. Clearly reversing pd' in the placement C_R results in a negative change in cost. This contradicts the remark that is given just before the theorem. Thus, R is optimal.

As for the running time of Algorithm *PathTotal*, Procedure *VertexAddition* takes $O(m)$ time and Procedure *MovePebbles* is implementable in $O(n)$ time using queues. Each of the Procedures *CostChange* and *Reverse* can be implemented in $O(mn)$ time using queues. These procedures are executed at most m times since at most m pebbles are reversed. Thus Procedure *PebbleDirection* terminates in $O(m^2n)$ time. Using the direction in which each pebble in P_D is to be shifted as determined by Procedure *PebbleDirection*, Procedure *PathSequence* can be used to generate a corresponding reconfiguration sequence in $O(mn)$ time as shown immediately following this proof.

Algorithm *PathTotal*, therefore, determines in $O(m^2n)$ time an optimal solution to the pebble shift problem when $G=(V,E)$ is a path. □

We now describe Procedure *PathSequence* that determines the actual reconfiguration sequence from the directions of the pebbles in P_D obtained by Procedure *PebbleDirection*.

Procedure *PathSequence*

1. Let R be an initially empty sequence of reconfiguration paths.
2. **while** there are still pebbles in P_D **do**
   A. Remove the first pebble pd from P_D.
   B. Let $d = iloc(pd)$.
   C. Remove from F_1 the vertex f that is closest to d in the $dir(pd)$ direction.
   D. Mark f as occupied and nondefective and d as vacant and defective.
   E. Add $Q^{dir(pd)}(d,f)$ to R.
   end while

end *PathSequence*

Total shift on a path. The same algorithm obtains the correct final placement when total shift is the cost criterion. Since no two paths which are in opposite directions intersect in the reconfiguration sequence obtained from Algorithm *PathTotal*, and Procedure *PebbleDirection* had minimized the total displacement, R is clearly an optimal reconfiguration sequence. The theorem pertaining to the correctness of the algorithm as well as its proof are identical to that provided for the total displacement and are not repeated. We now turn our attention to determining an algorithm for the case when $G=(V,E)$ is a cycle.

Total displacement on a cycle. We show how to modify Algorithm *PathTotal* for the case when $G=(V,E)$ is a cycle. Though Algorithm *PathTotal* with the modifications determines an optimal solution when $G=(V,E)$ is a cycle, our model for reconfiguration has some limitations. Due to these limitations it is possible to obtain a placement that maintains the relative ordering of the pebbles and has a lower total displacement than that of the placement generated by the reconfiguration sequence obtained by the algorithm. This is because displacement is not necessarily measured in the direction in which the pebble is moved. A detailed analysis of the limitations and solutions around them can be found in [6].

When $G=(V,E)$ is a cycle we refer to the clockwise direction as *RIGHT* and the counter clockwise direction as *LEFT*. The superscripts r and l in the notations used for reconfiguration paths represent the clockwise and counterclockwise directions, respectively, and all addition and subtraction operations on the indices of vertex labels are done modulo n. The reversal of a pebble is defined as before. Since G is a cycle, it is neither necessary nor possible to add vertices at an extreme end as in the case when G is a path.

To ensure the correctness of Lemma 1 we must order the list P_D properly after completing Phase I. We compute $CostChange(pd, LEFT)$ for each pebble pd in P_D in the placement obtained from Phase I. Let pd_i be that pebble for which $CostChange(pd_i, LEFT)$ returned the largest negative value. We reorder P_D as $[pd_i, pd_{i+1}, \ldots, pd_m, pd_1, pd_2, \ldots, pd_{i-1}]$. We show that once pd_i is reversed and located on vertex u, it will never be shifted out of u. Suppose that there another reconfiguration sequence R' that has a lower total displacement than the reconfiguration sequence obtained by the algorithm, and that applying R' located pd_i on a vertex $w \neq u$. In this case, after Phase I there must be some pebble pd_j whose reversal located pd_i on w such that $CostChange(pd_j, LEFT)$ had a greater negative value than $CostChange(pd_i, LEFT)$, a contradiction. Once pd_i has been reversed, the remainder of the problem can be treated just as if $G = (V, E)$ was a path with leftmost vertex u.

Total shift on a cycle. Algorithm *PathTotal* can be applied to minimize the total shift on a cycle as well. Here again we must reorder the pebbles in P_D as described in the minimization of total displacement on a cycle. The limitations of our model discussed with respect to total displacement on a cycle are not applicable here because by definition shift is measured along the path in which a pebble is moved.

Since the theorems and their proofs pertaining to the applicability of Algorithm *PathTotal* to minimize the total displacement on a cycle, and total shift on a path as well as a cycle are very much similar to Theorem 3 we have not stated nor proved them separately. We conclude this subsection by noting that though Algorithm *PathTotal* with minor modifications can be used to minimize cost functions based on maximum measures, it is possible to minimize these cost functions more efficiently as shown in the following.

Maximum displacement. We now present an $O(mn)$ time exact algorithm that solves the pebble shift problem with maximum displacement as the cost function when $G = (V, E)$ is a path. The same algorithm can be used when maximum shift is the cost function and $G = (V, E)$ is a simple path or a simple cycle. Though our algorithm can be used to minimize the total displacement when $G = (V, E)$ is a cycle, our model suffers from the same limitation that was discussed in the previously for total displacement when $G = (V, E)$ is a cycle.

The approach used in the algorithm is different from that of Algorithm *Path Total*. As before $D_1 = \{d_1, d_2, \ldots, d_m\}$ denotes the set of occupied and defective vertices in the initial placement, such that if d_k and d_{k+1} correspond to vertices v_i and v_j, respectively, then $i < j$. After setting the maximum displacement $md = 0$, the algorithm processes the occupied and defective vertices d of D_1 in increasing order of their indices. Let f_l and f_r be the closest vacant and nondefective vertices to the left and right of d, and P_l and P_r be the sets of pebbles that are located on vertices in the paths $Q^l(d, f_l)$ and $Q^r(d, f_r)$, respectively. Procedure *FindVacant* is used to locate the vertices f_l and f_r. If $p(d)$ is to be shifted left, each pebble in P_l must be shifted left, too. Likewise, if $p(d)$ is to be shifted to the right, each pebble in P_r must be shifted to the right.

Let md_l and md_r be the maximum of the displacements of the pebbles in P_l and P_r due to the reconfiguration along paths $Q^l(d, f_l)$ and $Q^r(d, f_r)$, respectively. The value of md_l is computed as follows. For a pebble p in P_l, vertices $iloc(p)$ and $cloc(p)$ represent, as before, its initial location and current location arrived at after reconfiguring along $Q^l(d, f_l)$, respectively. Furthermore, $dir(p)$ denotes the direction in which pebble p has to be shifted to reach $cloc(p)$ from $iloc(p)$. The displacement of pebble p is computed as $w(p) \times a^{dir(p)}(iloc(p), cloc(p))$. The maximum displacement md_l is computed as $\max_{p \in P_l} w(p) \times a^{dir(p)}(iloc(p), cloc(p))$ The maximum displacement md_r due to the shifting of pebble p to the right is similarly computed.

If $md_l < md_r$, we reconfigure along the path $Q^l(d, f_l)$, otherwise we reconfigure along the path $Q^r(d, f_r)$. This reconfiguration could change the direction of some pebbles in P_l or P_r. The values of $dir(p)$ for these pebbles p are updated accordingly. We then set $md = \max\{md, \min\{md_l, md_r\}\}$. In this manner Algorithm *PathMax* determines the directions in which all the pebbles must be shifted to obtain the final placement. As before, let P_D denote the list of pebbles $[pd_1, pd_2, \ldots, pd_m]$ that are located on vertices in D_l in the initial placement. Using Procedure *PathSequence* and the directions in which the pebbles in P_D are to be shifted, we obtain a reconfiguration sequence to solve the problem as in the case of Algorithm *PathTotal* in the previous section. The Procedure *MaxDisp* computes the values of md_l and md_r. Procedure *Reconfigure*(d, f_x), where x is either *LEFT* or *RIGHT*, performs a reconfiguration along path $Q^x(d, f_x)$ and updates the direction in which each pebble p must be shifted to reach $cloc(p)$ from $iloc(p)$.

We now describe Algorithm *PathMax* in detail. Procedures *FindVacant*, determines the closest vacant and nondefective vertex on the left or right of a given vertex. Procedure *MaxDisp* computes the maximum displacement for reconfiguring a given vertex to the left or the right and Procedure *Reconfigure* actually performs suc a reconfiguration. These procedures are quite simple and are not described in detail.

Algorithm (*PathMax*).

Input: A simple path $G = (V, E)$, a set of pebbles, their placement C_l on G with a set $D_l = \{d_1, d_2, \ldots, d_m\}$ of m occupied and defective vertices such that if d_k and d_{k+1} correspond to vertices v_i and v_j, respectively, then $i < j$, a set F_l of vacant and nondefective vertices, and the list $P_D = [pd_1, pd_2, \ldots, pd_m]$ of pebbles that are located on the vertices of D_l in the initial placement.

Output: An optimal reconfiguration sequence R.

1. $md = 0$. $D = D_l$.
2. **while** $D \neq \emptyset$ **do**
   A. Remove the lowest indexed vertex d from D.
   B. $f_l = FindVacant(d, LEFT)$
   C. $f_r = FindVacant(d, RIGHT)$
   D. **if** $f_l \neq NULL$ **then** $md_l = MaxDisp(d, LEFT)$ **else** $md_l = \infty$.
   E. **if** $f_r \neq NULL$ **then** $md_r = MaxDisp(d, RIGHT)$ **else** $md_r = \infty$.

F. **if** $md_l < md_r$ **then** $md_r = MaxDisp(d, RIGHT)$ **else** $md_r = \infty$.
G. **if** $md_l < md_r$ **then**
Reconfigure$(d, LEFT)$.
Update the direction of each pebble in P_l.
else
Reconfigure$(d, RIGHT)$
Update the direction of each pebble in P_r.
H. Mark d as vacant and defective.
I. $md = \max\{md, \min\{md_l, md_r\}\}$
end while
3. Using Procedure *PathSequence* described earlier and the directions of the pebbles in the list P_D, generate the reconfiguration sequence R.

end Algorithm *PathMax*.

Theorem 4. *Algorithm PathMax finds correctly in $O(mn)$ time an optimal solution to the pebble shift problem when $G = (V, E)$ is a simple path and the cost function to be minimized is maximum displacement.*

Proof. The term iteration in this proof refers to each execution of the **while** loop in Step 2 of Algorithm *PathMax*. After the completion of Iteration i, the occupied and defective vertices are all located on the path $Q^r(d_{i+1}, v_n)$. This is because (i) the pebble located on vertex d_i is shifted out of d_i and no pebble is moved into a vacant and defective vertex during the execution of iteration i and (ii) vertex d_i is marked as vacant and defective at the completion of iteration i.

Let $\mathscr{P}_i$ denote the pebble shift problem where $\{d_1, d_2, \ldots, d_i\}$ is the set of occupied and defective vertices in the initial placement. Note that since Procedure *Reconfigure*(d, x) performs a reconfiguration along the path $Q^x(d, f_x)$, on its completion in the ith iteration we have a placement that leaves no pebble on a defective vertex for problem $\mathscr{P}_i$. For notational convenience we refer to a placement obtained by applying an optimal reconfiguration sequence to the initial placement as an optimal solution in the following. We prove by induction on i that at the end of Iteration m, we have an optimal solution to the problem $\mathscr{P}_m$. Since the initial placement itself is an optimal solution to $\mathscr{P}_0$, the basis of the induction is true. The maximum displacement in this case is 0. For the induction hypothesis, we assume that after Iteration i we have an optimal solution to the problem $\mathscr{P}_i$. Let md denote the maximum displacement after this iteration. We now consider the problem $\mathscr{P}_{i+1}$, where $d_1, d_2, \ldots d_{i+1}$ are the occupied and defective vertices in the initial placement. Since all the vertices that are defective in the problem $\mathscr{P}_i$ are also defective in the problem $\mathscr{P}_{i+1}$, the maximum displacement for this problem is at least md. If either one of md_l and md_r is less than md, the solution obtained by the algorithm after Iteration $i+1$ still has a maximum displacement of md and is therefore an optimal solution for the problem $\mathscr{P}_{i+1}$.

Now suppose that both md_l and md_r are greater than md. After Iteration $i+1$ there is no vacant and nondefective vertex in the path $Q^r(f_l, f_r)$ other than one of f_l and f_r.

Let v_l' and v_r' be the vertices in $Q^r(f_l, f_r)$ that are adjacent to f_l and f_r, respectively, and let P' be the set of pebbles that are located on the vertices in the path $Q^r(f_l, f_r)$ just before iteration $i+1$. Note that after iteration i no pebble that was located in the initial placement on a vertex in the path $Q^r(f_l, f_r)$ has been shifted to a vertex not in the path, and that no pebble that was located in the initial placement on a vertex not on the path $Q^r(f_l, f_r)$ is shifted onto a vertex in the path. Let p_l and p_r be the pebbles in P' whose maximum displacements were computed to be md_l and md_r, respectively.

Suppose that there is a reconfiguration sequence R' whose application to the problem $\mathscr{P}_{i+1}$ leaves no pebble on a defective vertex and yields a maximum displacement of md' that is less than both md_l and md_r. There are three possible placements that may be generated by the application of R'. (1) All the pebbles in P' are located to the left of v_r'. (2) All the pebbles in P' are located to the right of v_l', and (3) Some pebbles in P are shifted to the left of v_l and some to the right of v_r, leaving a vacant and nondefective vertex other than f_l and f_r in the path $Q^r(f_l, f_r)$. In the first placement, p_l must be shifted further to the left than in the case after iteration $i+1$. Thus $md' > md_l > \min\{md_l, md_r\}$. Similarly, in the second placement, $md' > md_r > \min\{md_l, md_r\}$. Finally, in the third placement, either p_l is shifted further to the left or p_r is shifted further to the right than in the case after iteration $i+1$. In either case $md' > \min\{md_l, md_r\}$. Hence there is no reconfiguration sequence for the problem $\mathscr{P}_{i+1}$ for which the maximum displacement is less than $\min\{md_l, md_r\}$, proving that after iteration $i+1$ we obtain an optimal solution for $\mathscr{P}_{i+1}$.

Each iteration takes $O(n)$ steps and at most m such iterations are executed. Hence, the **while** loop of Algorithm *PathMax* executes in $O(mn)$ time. Procedure *PathSequence* takes $O(mn)$ time to execute as shown earlier. Therefore, Algorithm *PathMax* executes in $O(mn)$ time. □

Maximum shift on a path. In the sequence of reconfiguration paths generated by Algorithm *PathMax*, no pebble has been shifted both in the right and left directions to obtain the final placement. This implies that the maximum displacement and the maximum shift have the same value. Therefore, Algorithm *PathMax* can be used to obtain an optimal solution to the pebble shift problem when $G=(V,E)$ is a simple path and maximum shift is the cost function.

Maximum displacement on a cycle. Algorithm *PathMax* can also be used to minimize the maximum displacement when $G=(V,E)$ is a simple cycle. Since the displacement of a pebble is not necessarily measured in the direction in which the pebble was moved our model faces the same limitations that were mentioned with respect to total displacement.

Maximum shift on a cycle. As mentioned above, Algorithm *PathMax* determines an optimal reconfiguration sequence when the cost function is maximum displacement and the graph $G=(V,E)$ is a simple cycle.

Having analyzed all the variations of the pebble shift problem on both paths and cycles, in the following sections we turn our attention to general graphs. We first provide an optimal solution for the case when all the pebbles are of equal weight.

3.2. Unweighted Pebble Shift

The total displacement minimization problem can be formulated as a bipartite matching problem [9] and solved in $O(n^3)$ time when all the pebbles are of equal weight. Narasimhan et al. [7] have formulated the total shift minimization problem in the same way. For the sake of completeness we repeat their algorithm in this section. Recently, Codenotti and Tammassia [1] have addressed the reconfiguration of programmable arrays. As it turns out their formulation is a special case of the pebble shift problem which minimizes the maximum displacement when all the pebbles are of the same weight. They have provided an exact algorithm that runs in $O(mn^3)$ time for this problem.

Total displacement. We start by constructing a new graph $B=(U,A)$ from the graph $G=(V,E)$ and the initial placement of pebbles on its vertices as follows. Let U_1 and U_2 be the sets of vertices that correspond to the occupied and defective and the vacant and nondefective vertices, respectively, of V. If $|U_1| > |U_2|$, the problem has no solution; otherwise we set $U = U_1 \cup U_2$ and $A = \{(u,u') | u \in U_1 \text{ and } u' \in U_2\}$. For each edge $(u,u') \in A$, we set the value of its length $l(u,u')$ to be the length of a shortest path from v to v' in G that correspond to the vertices u and u', respectively.

Since each edge in A has one end vertex in U_1 and another end vertex in U_2 and each vertex in U_1 is connected to every vertex in U_2 by an edge in A, the graph $B=(U,A)$ is a so-called *complete bipartite graph*. For such a graph, a *matching* is a set of edges $M \subseteq A$ no two of which have the same end vertex. Its cost is defined as $\sum_{(u,u')\in M} l(u,u')$. A matching with the maximum cardinality is called a *maximum matching*. Since B is a complete bipartite graph with $|U_1| \leqslant |U_2|$, such a matching is of cardinality $|U_1|$. A maximum matching that has the lowest cost is called a *minimum cost maximum matching*. Such a matching can be found in $O(|U|^3)$ time [9].

Let $(u_i,u_j) \in A$ be an edge in a maximum matching M of B where u_i and u_j represent vertices v_i and v_j in V of G, respectively. We remove this edge from M and perform a reconfiguration along a shortest path $Q(v_i,v_j)$ in G as follows. Let $v_1 \in V$ be the vacant and nondefective vertex that is closest to v_i on $Q(v_i,v_j)$ and let it be represented by $u_1 \in U_2$ in B. It is possible that $v_1 = v_j$. Let the path $Q(v_i,v_1)$ be that portion of $Q(v_i,v_j)$ that starts at v_i and terminates at v_1. We reconfigure along the path $Q(v_i,v_1)$ in G. If $u_1 \neq u_j$, since M is a matching of cardinality $|U_1|$, there must be a vertex $u_k \neq u_i$ in U_1 such that $(u_k,u_1) \in M$. Furthermore, there is an edge $(u_k,u_j) \in A$. We replace the edges (u_k,u_1) with (u_k,u_j) in M.

We now present an outline of our algorithm for the total displacement minimization problem when all the pebbles are of the same weight. We call it the *Unweighted Reconfiguration Algorithm* (*UNWGT*).

Unweighted Reconfiguration Algorithm (UNWGT)

Input: A graph $G=(V,E)$, a set of pebbles their initial placement on G.

Output: A reconfigured placement of the pebbles on nondefective vertices of G.

1. Using the initial placement of pebbles on the graph $G=(V,E)$, create the bipartite graph $B=(U,A)$ with $U=U_1 \cup U_2$ as described above.
2. **if** $|U_1| > |U_2|$ **then** report failure and **exit**.
3. Obtain a minimum cost maximum matching M of B.
4. **while** M is not empty **do**
   a. Pick an edge $(u_i,u_j) \in M$ and find its corresponding reconfiguration path $Q(v_i,v_l)$ in G as described above.
   b. If $u_l \neq u_j$, find the edge $(u_k,u_l) \in M$ and replace it with the edge (u_k,u_j) in M.
   c. Set $M=M-\{(u_i,u_j)\}$. Mark v_i as vacant and defective and v_l as occupied and nondefective.
5. Output the reconfigured placement of the pebbles on nondefective vertices of G.

The following theorem is easily established.

Theorem 5. *Algorithm UNWGT correctly solves the total displacement minimization problem for any graph $G=(V,E)$ in $O(n^3)$ time, with a minimum total displacement when all the pebbles are of the same weight.*

Proof. Since all pebble weights are equal, the cost of shifting pebbles is only dependent upon the total length of all reconfiguration paths in a solution to the pebble shift problem. Since a minimum cost maximum matching M of the bipartite graph $B=(U,A)$ is found in $O(|U|^3)$ time [9] and $|U| \leqslant n$, Step 3 can be executed in $O(n^3)$ time. The cost of M is the sum of the lengths of edges of M. Each such edge represents a reconfiguration path in G. Thus the sum of the lengths of such paths is also a minimum. Therefore, if $u_l=u_j$ at each execution of Step 4.a., we have a reconfiguration sequence with the minimum total displacement at the termination of the algorithm.

Suppose that $u_l \neq u_j$ in some iteration of Step 4. Let the path $Q(v_l,v_j)$ be that portion of $Q(v_i,v_j)$ that excludes its initial portion covered by $Q(v_i,v_l)$. When $u_j \neq u_l$, the edges (u_i,u_j) and (u_k,u_l) belong to the matching M and their contribution to the cost of M is written as $a(v_i,v_j)+a(v_k,v_l)=a(v_i,v_l)+a(v_l,v_j)+a(v_k,v_l)=a(v_i,v_l)+a(v_k,v_j)$. Note that the last term is the length of the path $Q(v_k,v_j)$, which is composed of the paths $Q(v_k,v_l)$ and $Q(v_l,v_j)$. Therefore, the modifications of the reconfiguration path and the minimum cost maximum matching M at Steps 4a and b do not change the optimality of the reconfiguration sequence to be obtained at the termination of the algorithm.

Since $|M|=|U_1|$, it takes $O(|U_1| \times n)$ time to complete Step 4, and $|U_1| \leqslant n$. Therefore, the algorithm correctly solves the unweighted pebble shift problem in $O(n^3)$ time. □

When we apply the above algorithm for the reconfiguration of a programmable array, the construction of graphs $G=(V,E)$ and $B=(U,A)$ is rather simple. The graph G

simply reflects the physical arrangements of the PEs of the array and the LEs of the circuit. For example, assuming that routing is allowed only along horizontal and vertical channels, as is commonly the case in such arrays, G becomes a grid graph. In this case the length of each edge (u,u') in the bipartite graph B, is computed to be the Manhattan distance from v to v' in G.

The approach proposed by Codenotti and Tammassia [1] minimizes the maximum displacement for the special case of our problem in which all the pebbles are of equal weight and all the edges are of equal length. Interestingly enough, with minor modifications the approach can also be used when edge lengths are arbitrary. The NP-Completeness results presented in the next section indicate that it is very unlikely that a polynomial solution exists for the general case of the pebble shift problem and hence we resort to heuristic algorithms presented by Narasimhan et al. [8] for the general case.

4. NP-completeness

In this section we first prove that *TDP* is NP-complete for undirected as well as directed graphs even when all edges are of the same length and all pebbles are of weight 1 or 2. We use a polynomial transformation from the following problem called Exact Cover by Three Sets ($X3C$) to *TDP*.

Exact Cover by Three Sets ($X3C$)

Instance: A set X with $|X| = 3q$ and a collection S of 3-element subsets of X.

Question: Does S contain an exact cover for X, i.e., a subcollection $S' \subseteq S$ such that every element of X occurs in exactly one member of S' ?

It is well known that $X3C$ is NP-complete [3]. Using a slightly different transformation we then prove that *MDP* is NP-complete for directed acyclic graphs even when all edges are of the same length and all pebbles are of weight 1 or 2.

Theorem 1. *TDP is NP-complete for an undirected graph* $G=(V,E)$ *with* $l(v_i,v_j)=1$ *for each* $(v_i,v_j) \in E$ *and* $w(p) \in \{1,2\}$ *for each* $p \in P$.

Proof. Clearly, *TDP* is in NP. Let $X = \{x_1, x_2, \ldots, x_{3q}\}$ and $S = \{s_1, s_2, \ldots, s_m\}$ be an instance of $X3C$. To show a polynomial transformation from $X3C$ to *TDP* we construct from X and S a graph $G=(V,E)$, a set P of pebbles, their initial placement, and an integer constant c as follows. For each element $x_i \in X$ we define a defective vertex d_i. For each subset $s_j \in S$ we create a subgraph $G_j=(V_j,E_j)$, where $V_j=\{v_{j1}, v_{j2}, f_{j1}, f_{j2}, f_{j3}\}$ is a set of nondefective vertices and $E_j = \{(v_{j1}, v_{j2})\} \cup \{(v_{j2}, f_{jr}) | r = 1,2,3\}$. We call G_j a *subset component* for s_j. Let $E^* = \{(d_i, v_{j1}) | x_i \in s_j,\ j = 1,2,\ldots,m\}$. Let $D_1 = \{d_i | i = 1,2,\ldots,3q\}$ and $F_1 = \{f_{jr} | j = 1,2,\ldots,m; r = 1,2,3\}$. We now define $V = D_1 \cup (\bigcup_{j=1}^{m} V_j)$ and $E = E^* \cup (\bigcup_{j=1}^{m} E_j)$. All the edges in E are of length 1.

Let P contain $3q+m$ pebbles of weight 1 and m pebbles of weight 2. For the initial placement we place a pebble of weight 1 on each vertex in $D_1 \cup \{v_{j1} | j=1,2,\ldots,m\}$ and a pebble of weight 2 on each vertex in $\{v_{j2} | j = 1,2,\ldots,m\}$. Fig. 5 shows an instance

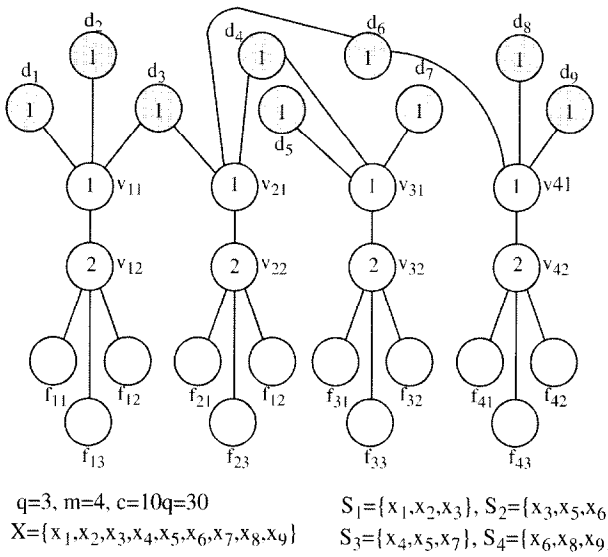

Fig. 5. An example of a transformation from $X3C$ to TDP.

of $X3C$ and its corresponding graph $G=(V,E)$ and the initial placement of pebbles. Finally we set $c=10q$. The above transformation can easily be done in polynomial time.

Suppose that $S' \subseteq S$ is a solution to $X3C$. Corresponding to each $s_j=\{x_{j1},x_{j2},x_{j3}\} \in S'$ we use the shortest paths from d_{j1},d_{j2}, and d_{j3} to f_{j1},f_{j2}, and f_{j3}, respectively, as reconfiguration paths, in this order. After this reconfiguration, the pebbles from the vertices $v_{j2},v_{j1},d_{j1},d_{j2}$, and d_{j3} are moved to the vertices $f_{j1},f_{j2},f_{j3},v_{j2}$, and v_{j1}, respectively. The total displacement is $q \times (2+2+3+2+1)=10q$, which is $\leqslant c$. Thus there is a solution to TDP.

On the other hand, if TDP has a solution, there exists a reconfiguration sequence RS with $TD(RS) \leqslant 10q$. We say that a subset component $G_j=(V_j,E_j)$ for s_j is *entered* r times if the number of vacant and nondefective vertices in the set $\{f_{j1},f_{j2},f_{j3}\}$ that belong to some reconfiguration path in the solution is r. From the definition of a reconfiguration path and the construction of the graph we note that each pebble of weight 2 can be moved only once. When a subset component is entered exactly once, the displacement is $2+1+1=4$ which is obtained by using any one of the paths in $\{[d_{ji},v_{j1},v_{j2},f_{jr}]|i,r=1,2,3\}$. Likewise when a subset component is entered exactly twice and three times, the displacements are 7 and 10, respectively. Suppose that y_1,y_2, and y_3 are the numbers of subset components that are entered once, twice, and three times, respectively. Since $|D_1|=3q$, we note that $y_1+2y_2+3y_3=3q$. The total displacement due to these reconfiguration paths is $4y_1+7y_2+10y_3$, which must be $\leqslant 10q$. Combining these two equations, we obtain $2y_1+y_2 \leqslant 0$, which is possible only when $y_1=y_2=0$. Therefore, $y_3=q$. This means that either a subset component is entered three times or not at all. Furthermore, each of the $3q$ reconfiguration paths

must begin at some d_i and enter a subset component. By choosing those subsets $s_k \in S$ whose corresponding subset components are entered in the solution to *TDP*, we obtain a solution to $X3C$. □

Remark. By assigning appropriate directions to the edges, it is easy to see that *TDP* is NP-complete even when $G=(V,E)$ is a directed acyclic graph.

The NP-completeness result can be easily extended to cubic planar undirected graphs as well as cubic planar acyclic directed graphs. We have not included these extensions in this paper since they do not contribute any further insight into the problem.

Theorem 2. *MDP is NP-complete for a directed acyclic graph* $G=(V,E)$ *with* $l(v_i,v_j)=1$ *for each* $(v_i,v_j)\in E$ *and* $w(p)\in\{1,2\}$ *for each* $p\in P$.

Proof. Clearly *MDP* is in the class NP. Let $X=\{x_1,x_2,\ldots,x_{3q}\}$ and $S=\{s_1,s_2,\ldots,s_m\}$ be an instance of $X3C$. Corresponding to X and S we construct a directed acyclic graph $G=(V,E)$, a set P of pebbles, their initial placement, and an integer c as follows. For each element $x_i\in X$ we define a defective vertex d_i. For each subset $s_j\in S$ we create a subgraph $G_j=(V_j,E_j)$, where $V_j=\{v_j,f_{j1},f_{j2}\}\cup\{u_{ji}|i=1,2,\ldots,q+1\}$ and $E_j=\{(v_j,u_{j1})\}\cup\{(u_{ji},u_{ji+1})|i=1,2,\ldots,q\}\cup\{(u_{jq+1},f_{j1}),(u_{jq+1},f_{j2})\}$. We call G_j a *subset component* for s_j. In each such component G_j the vertices v_j, f_{j1} and f_{j2} are labeled as nondefective and the remaining vertices u_{ji}s are labeled as defective. We then create a special subgraph $G'=(V',E')$, where $V'=\{v'_i|i=1,2,\ldots,m+q\}$ is a set of nondefective vertices and $E'=\{(v'_i,v'_{i+1})|i=1,2,\ldots,m+q-1\}$. Let $E^*=\{(d_i,v_j)|x_i\in s_j,j=1,2,\ldots,m\}$ and $E^{**}=\{(u_{jq},v'_j)|j=1,2,\ldots,m\}$. Let $D_1=\{d_i|i=1,2\ldots,3q\}$ and $F_1=\{f_{j1},f_{j2}|j=1,2,\ldots,m\}\cup\{v'_{m+i}|i=1,2,\ldots,q\}$. We now define $V=D_1\cup V'\cup(\bigcup_{j=1}^{m}V_j)$ and $E=E^*\cup E^{**}\cup E'\cup(\bigcup_{j=1}^{m}E_j)$. All the edges are of length 1. The graph $G=(V,E)$ is clearly a directed acyclic graph.

Let P contain $3q+m$ pebbles of weight 1 and m pebbles of weight 2. For the initial placement we place a pebble of weight 1 on each vertex in $D_1\cup\{v'_j|j=1,2,\ldots,m\}$ and a pebble of weight 2 on each vertex in $\{v_j|j=1,2,\ldots,m\}$. Fig. 6 shows an instance of $X3C$ and its corresponding graph $G=(V,E)$ and initial placement of pebbles. Finally, we set $c=2q+3$. The above transformation can easily be done in polynomial time.

Suppose that $S'\subseteq S$ is a solution to $X3C$. We assume, without any loss of generality, that the elements of S' are arranged in increasing order of their indices. In such an order, we choose subsets in S' one by one and perform the following reconfiguration operations. Let $s_j=\{x_{j1},x_{j2},x_{j3}\}$ be the rth subset in S'. We first reconfigure along the shortest path from d_{j3} to v'_{m+r}. This will move the pebble of weight 2 from vertices v_j to v'_j. We then reconfigure along the shortest paths from d_{j1} and d_{j2} to f_{j1} and f_{j2}, respectively. The displacement of each pebble of weight 2 is $2\times(q+1)$. The pebbles of weight 1 initially on d_{j3} and d_{j1} are displaced by $q+3$ and that on d_{j2} is displaced by 1. The pebbles of weight 1 initially on v'_js for $j=1,2,\ldots,m$ are displaced by at most q. Since $q\geqslant 0$, the maximum displacement is $\max\{2q+2,q+3,1,q\}<c$. Thus there is a solution to *MDP*.

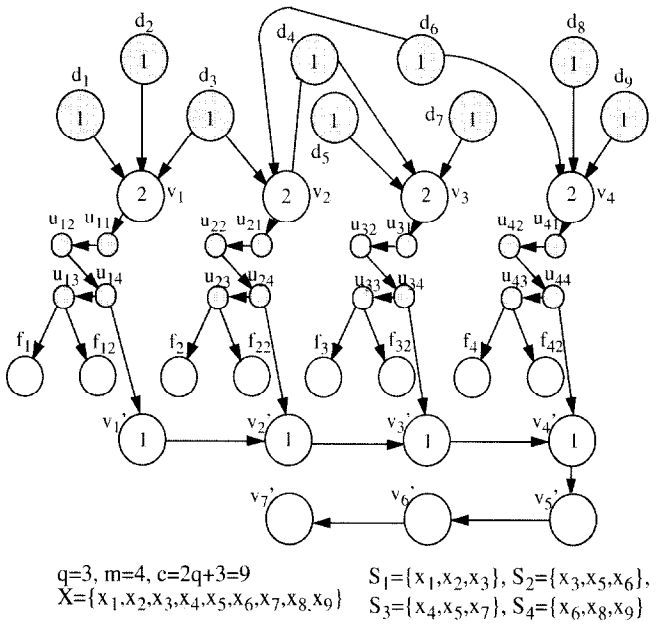

Fig. 6. An example of a transformation from *X3C* to *MDP*.

On the other hand, suppose that *MDP* has a solution. From the construction of G, we note that every reconfiguration path must pass through some vertex v_j in a subset component $G_j = (V_j, E_j)$ for $s_j \in S$. A pebble of weight 2 can only be moved to a vertex in V', otherwise it will be moved across a distance of $q+2$ for a displacement of $2q+4$, which is $> c$. Suppose that pebbles of weight 2 are moved to vertices in V' from their initial locations of v_k and v_l, where $k < l$. To ensure that the pebble initially on v_l is not displaced by more than c, it must be moved after the pebble that was initially on v_k. Since $|V' \cap F_1| = q$, at most q pebbles of weight 2 can be moved to vertices in V'. Once a pebble of weight 2 is moved out of v_j, exactly two more paths ending at the vertices f_{j1} and f_{j2} can be used for reconfiguration without the displacement of a pebble exceeding c. Therefore, there are at most $3q$ paths available for reconfiguration. Since $|D_1| = 3q$, the reconfiguration sequence must contain all these paths. As mentioned above, these $3q$ paths must pass through the q vertices v_js. Furthermore, exactly three such paths can pass through each v_j. Therefore, each v_j must belong to three reconfiguration paths or no path at all. By choosing the subsets each of which corresponds to three reconfiguration paths in the solution to *MDP*, we obtain a solution to *X3C*. □

Remark. As in the case of *TDP* the NP-completeness result for *MDP* can be easily extended to cubic planar acyclic directed graphs. Once again we have not included these extensions in this paper since they do not contribute any further insight into the problem.

Table 1
Summary of complexity results for the pebble shift problem

Cost measure	Pebble weights 1 or 2		Arbitrary pebble weights and G is a cycle/path	Equally weighted pebbles
	Undirected cubic planar	Directed acyclic cubic planar		
Total disp.	$\mathcal{NP}$-C	$\mathcal{NP}$-C	$O(m^2n)$	$O(n^3)$
Total shift	$\mathcal{NP}$-C	$\mathcal{NP}$-C	$O(m^2n)$	$O(n^3)$
Max. disp.	??	$\mathcal{NP}$-C	$O(mn)$	$O(mn^3)$ [1]
Max. shift	??	$\mathcal{NP}$-C	$O(mn)$	$O(mn^3)$ [1]

5. Conclusions

In this paper we have addressed the pebble shift problem that was originally defined by Narasimhan et al. based on their model for reconfiguration of programmable arrays. We have formally defined the problem and four different cost functions that are associated with it. For the case when the graph $G=(V,E)$ is a path, we have presented an $O(m^2n)$ time exact algorithm for costs based on total measures and an $O(mn)$ time exact algorithm when the costs are based on maximum measures where n is the total number of vertices in the graph and m is the number of occupied and defective vertices in the initial placement of the pebbles on the graph. We have then presented an $O(n^3)$ time algorithm for the case when $G=(V,E)$ is arbitrary and all the pebbles are of the same weight. Finally we have proved that the problem is NP-hard for costs based on total measures even when the graph $G=(V,E)$ is a undirected cubic planar or directed acyclic cubic planar and each pebble is of weight 1 or 2. We have also proved that the problem is NP-hard for costs based on maximum measures is NP-hard when the graph $G=(V,E)$ is directed acyclic cubic planar and when the pebbles are all of weight 1 or 2. Table 1 summarizes our results for the pebble shift problem.

Acknowledgements

We thank Professor R. Greenberg of the Department of Mathematical and Computer Sciences, Loyola University (formerly of the Department of Electrical Engineering, University of Maryland) for bringing to our attention an error in our earlier pebble shift algorithm for paths and cycles and for providing useful discussions on the new algorithm.

References

[1] B. Codenotti, R. Tammassia, A network flow approach to the reconfiguration of VLSI arrays, IEEE Trans. Comput., 40 (1) (1991) 118–121.
[2] M.E. Dyer, A.M. Frieze, Planar 3DM is NP-complete, J. Algorithms 7 (1986) 174–184.

[3] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to The Theory of NP-completeness, Freeman, San Francisco, CA, pp. 199, 1979.
[4] S. Kirkpatrick, C.D. Gelatt, P.M. Vecchi, Optimization by simulated annealing, Science 220, (4598) (1983) 671–680.
[5] V.P. Kumar, A.T. Dahbura, F.H. Fischer, P.M. Joula, An approach for the yield enhancement of Gate arrays, Proc. IEEE Int. Conf. on Computer-Aided Design, Santa Clara CA, Nov. 1989, pp. 226–229.
[6] J. Narasimhan, K. Nakajima, C.S. Rim, A.T. Dahbura, Reconfiguration for programmable ASIC arrays, Technical Report, UMIACS-TR-92-7, UMIACS, University of Maryland, College Park, MD 1992.
[7] J. Narasimhan, C.S. Rim, K. Nakajima, A.T. Dahbura, Yield enhancement of programmable chips by reconfiguration, Proc. IEEE Int. Conf. on Wafer Scale Integration, San Francisco, CA, Jan. 1991, pp. 178–184.
[8] J. Narasimhan, K. Nakajima, C.S. Rim, A.T. Dahbura, Yield enhancement of programmable ASIC arrays by reconfiguration of circuit placements, IEEE Trans. Computer-Aided Des. Integrated Circuits Systems 13 (8) (1994) 976–986.
[9] C.H. Papadimitrou, K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Prentice-Hall, Englewood Cliffs, NJ, 1982, pp. 395–396.
[10] S.K. Tewksbury, Wafer-Level Integrated Systems: Implementation Issues, Ch. 9 Kluwer Academic Publishers, Boston, MA, 1989.